Chapter 3

Methodology

In this chapter we present our methodology to answer the research questions enunciated in **??**. We investigate three research questions. In the first question, we artificially generate WebAssembly program variant, and we qualitatively compare the ability of CROW, on of the contribution of this thesis, to generate statically different, yet functionally equivalent variants. Our second research question compares the bahavior of artifically created variants through traces, execution time and preservation in machine code. The final research question evaluates the feasibility of using the program variants in security sensitive environments such as Edge-Cloud computing proposing a multivariant execution approach.

We first, enunciate the corpora of programs used to answer the research questions. We establish the metrics, and we set the configuration for the experiments. Besides, we describe the protocol for each research question.

3.1 Corpora

We answer the research questions with three corpora of programs appropriate for our experiments. In Table 3.1 we listed the corpus name, the number of programs inside the corpus, the total number of functions, the range of lines of code and the original location of the corpus. In the following, we describe the filtering and description of each corpus.

1. **Rosetta** corpus is part of the CROW contribution []. We take programs from the Rosetta Code project[1]. This website hosts a curated set of solutions for specific programming tasks in various programming languages. It contains a wide range of tasks, from simple ones, such as adding two numbers, to complex algorithms like a compiler lexer. We first collect all C programs from Rosetta Code, which represents 989 programs as of 01/26/2020. We then apply a number of filters: the programs should successfully compile, they should not

[1] `http://www.rosettacode.org/wiki/Rosetta_Code`

require user inputs, the programs should terminate and should not provide in non-deterministic results. The result of the filtering is a corpus of 303 C programs. All programs have a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.

2. **Libsodium** is part of both CROW and MEWE contributions [] []. This project is an encryption, decryption, signature and password hashing library which can be ported to WebAssembly. We selected 5 programs or endpoints to answer our research questions. These endpoints have between 8 and 2703 lines of code per function. The project is selected based on their suitability for diversity synthesis with CROW, *i.e.,* the project should have the ability to collect its modules in LLVM intermediate representation and the project should be easily portable Wasm/WASI.

3. **QrCode** is part of the MEWE contribution. This project is a QrCode and MicroQrCode generator written in Rust. We selected 2 programs or endpoints to answer our research questions. These endpoints have between 4 and 725 lines of code per function. As Libsodium, we select this project due to its suitability for diversity synthesis with CROW.

Corpus name	No. programs	No. functions	LOC range	Location
Rosetta	303	303	7 - 150	`http://rosettacode.org/wiki/Rosetta_Code`
Libsodium	5	869	8 - 2703	`https://github.com/jedisct1/libsodium`
QrCode	2	1849	4 - 725	`https://github.com/kennytm/qrcode-rust`
Total	310	3021		

Table 3.1: Corpora description. The table is composed by the name of the corpus, the number of programs, the number of functions, the lines of code range and the location of the corpus.

3.2 RQ1. To what extent we can generate program variants for WebAssembly.

In this research question, we investigate whether we can artifically generate program variants for WebAssembly. We use CROW to generate program variants from an

original program, written in C/C++ or directly passing a LLVM bitcode module to it. We analyze the properties the programs should have to generate a handful number of variants.

3.2.1 Metrics

To assess our approach's ability to generate WebAssembly binaries that are statically different, we compute the number of unique variants generated by CROW for each original function. For each program and its variants we calculate the count of unique generated variants.

Metric 1 *Population size $S(P)$: Given a program P and its generated variants V, the population size metric is defined as.*

$$S(P, V) = |V \cap \{P\}|$$

3.2.2 Protocol

The generation of program variants is part of the answering of all our research questions. For such a reason the first step and research question is related to how we can accomplish such task. To generate variants, we pass each of the **TODO** XX functions to CROW. CROW's workflow synthesizes program variants with an enumerative strategy. All possible programs that can be generated for a given language (LLVM in the case) are constructed and verified for equivalence. There are two parameters to control the size of the search space and hence the time required to traverse it. On the one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions used for the synthesis. On the other hand, in our experiments, we use between 1 instruction (only additions) and 60 instructions (all supported instructions in the synthesizer).

These two configuration parameters allow the user to find a trade-off between the number of variants that are synthesized and the time taken to produce them. In Table 3.2 we listed the configuration for both corpora. For the current evaluation, given the size of the corpus, we set the exploration time to 1 hour maximum per function for Rosetta . In the cases of Libsodium and QrCode, we set the timeout to 5 minutes per function in the exploration stage. We set all 60 supported instructions in CROW for Rosetta , Libsodium and QrCode corpora. We then collect the generated program variants for each function, and we calculate the Metric 1 for them.

CORPUS Name	Exploration timeout	Max. instructions
Rosetta	1h	60
Libsodium	5m	60
QrCode	5m	60

Table 3.2: CROW tweaking for variants generation. The table is composed by the name of the corpus, the timeout parameter and the count of allowed instructions during the synthesis process.

3.3 RQ2. To what extent the artifically generated variants are different?

In this second research question, we investigate to what extent the artificially created variants are different between them and to the original program. To conduct this research question, we perform three experiments: static comparison, dynamic comparison and variant preservation. The experiments are performed over the original programs from corpora described in section 3.1 and their variants generated in the answering of section 3.2. We take an original program and their variants composing pairs of programs and we do a fine-grained comparison.

3.3.1 Metrics

To two programs we propose use a global alignment approach. In a previous work of us, we highlighted how this approach can be used to measure similarity between program traces [?] . In the following we define the *dt_static* metric.

Metric 2 *dt_static: Given two programs P_X and V_X written in X code, dt_static(P_X, V_X), computes the DTW distance between the corresponding program instructions for representation X.*

A dt_static(P_X, V_X) of 0 means that the code of both the original program and the variant is the same, i.e., they are statically identical in the representation X. The higher the value of dt_static, the more different the programs are in representation X.

Notice that for comparing WebAssembly programs and its variants, the metric is the instantiation of dt_static with $X = WebAssembly$.

We measure the difference between programs at runtime by evaluating their execution trace, at function and instruction level. **TODO** Replace and explain the stack trace as stack operations **TODO** Add and example

Metric 3 *dt_dyn: Given a program P, a CROW generated variant P' and T a trace space (*$T \in \{Function, Instruction\}$*) dt_dyn(P,P',T), computes the DTW distance between the traces collected during their execution in the T space. A dt_dyn of* 0 *means that both traces are identical.*

The higher the value, the more different the traces.

Also, we include the measuring of the execution time of the programs.

Metric 4 *Execution time: Given a WebAssembly program P, the execution time is the time spent to execute the binary.*

WebAssembly is an intermediate language and compilers use it to produce machine code. For program variants, this means that compilers can undo artificial introduced transformations, for example, through optimization passes. When a code transformation is maintained from the first time it is introduced to the final machine code generation is a preserved variant. For this mentioned reasoning we need a preservation metric. The key property we consider is as follows:

Property 1 *Preservation: Given a program P and a CROW generated variant P', if dt_static(*P_{Wasm}*,* P'_{Wasm}*) > 0 and dt_static(*P_{x86}*,* P'_{x86}*) > 0* $\implies$ *both programs are still different when compiled to machine code.*

If the property fits for two programs, then the underlying compiler does not remove the transformations made by CROW. Notice that, this property only makes sense between variants of the same program, including the original.

Metric 5 *Preservation: Given a WebAssembly programs P and a collection of generated variants V, the preservation ratio is the number of pair of programs that fit with Property 1 over the total number of program pairs.*

3.3.2 Protocol

For each function on the corpora, we compare the sequence of instructions of each variant with the initial program and the other variants. We obtain the **??** values for each program-variant WebAssembly pair code. We compute the DTW distances with STRAC [**?**].

To compare program and variants behavior during runtime, we analyze all the unique program variants generated by CROW in a pairwise comparison as well. We use SWAM[2] to collect the function and instruction traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions including the stack operations. We compute the DTW distances with STRAC [**?**].

[2] `https://github.com/satabin/swam`

Furthermore, we collect the execution time, Metric 4, for all programs and their variants. We execute each program or variant TODO XXX times, and we compare the collected execution times using a Mann-Withney test [?] in a pairwise strategy.

We collect Metric 5 for all programs and their generated variants. We use the engines listed in Table 3.3.We only take into account the x86 representation after the WebAssembly code is compiled to the machine code. This decision is not arbitrary, according to the study of TODO Paper on binary diff survey , any conclusion carried out by comparing two program binaries under a specific target can be extrapolated to another target for the same binaries.

Name	Properties
V8 [?]	V8 is the engine used by Chrome and NodeJS to execute JavaScript and WebAssembly. TODO Explain compilation process
wasmtime [?]	Wasmtime is a standalone runtime for WebAssembly. This engine is used by the Fastly platform to provide Edge-Cloud computing services. TODO Explain compilation process

Table 3.3: Wasm engines used during the diversification assessment study. The table is composed by the name of the engine and the description of the compilation process for them.

3.4 RQ3. To what extent the artificially generated variants can be used to enforce security?

In the last research question we study whether the created variants can be used in real-world applications. For this purpose, we build multivariant binaries to be deployed at the Edge in the Fastly platform. We build multivariant binaries using the variants generated for the programs of the Libsodiumand QrCodecorpora, $2+5$ programs, for which we analyze their involved functions and the generated variants.

We assess the ability of MEWE to produce binaries that actually exhibit random execution paths when executed on one edge node. We check the diversity of execution traces gathered from the execution of a multivariant binary. The traces are collected from all edge nodes in order to assess MVE at a worldwide scale. MEWE generates binaries that embed a multivariant behavior. We measure to what extent MEWE generates different execution times on the edge. Then, we discuss how multivariant binaries contribute to less predictable timing side-channels.

3.4.1 Metrics

3.4.2 Setup

Appended papers